24장 클로저

- 클로저

  함수와 그 함수가 선언된 렉시컬 환경과의 조합

```
const x = 1;

function outerFunc() {
  const x = 10;

  function innerFunc() {
    console.log(x); // 10
  }

  innerFunc();
}

outerFunc();
```

- 중첩 함수 innerFunc 내부에서 자신을 포함하고 있는 외부 함수 outerFunc의 변수 x에 접근 가능
- 만약 innterFunc 함수가 outerFunc 함수 내부에서 정의된 중첩 함수가 아니라면,

  내부에서 호출한다고 하더라도 변수 x에 접근이 불가능

24.1 렉시컬 스코프

- 렉시컬 스코프 (Lexical scope)

  자바스크립트 엔진은 함수가 정의된 위치에 따라 상위 스코프를 결정한다.

```
const x = 1;

function foo() {
  const x = 10;
```

```
  bar();
}

function bar() {
  console.log(x);
}

foo(); // ?
bar(); // ?
```

- 함수의 상위 스코프를 결정한다는 말은 곧, **"렉시컬 환경의 외부 렉시컬 환경에 대한 참조에 저장할 참조값을 결정한다."** 는 말과 같다.
- 따라서 렉시컬 스코프란,

렉시컬 환경의 "외부 렉시컬 환경에 대한 참조"에 저장할 참조값 (상위 스코프에 대한 참조)는

함수 정의가 평가되는 시점에 함수가 정의된 환경 (위치)에 의해 결정된다.

24.2 함수 객체의 내부 슬롯 [[Environment]]

- 함수는 자신의 내부 슬롯 [[Environment]]에 자신이 정의된 환경, 즉 상위 스코프의 참조를 저장
- 이 내부 슬롯에 저장된 현재 실행 중인 실행 컨텍스트의 렉시컬 환경의 참조가 바로 상위 스코프
- 동시에 자신이 호출되었을 때 생성될 함수 렉시컬 환경의 "외부 렉시컬 환경에 대한 참조"에

  저장될 참조값.
- 함수 객체는 내부 슬롯 [[Environment]]에 저장한 렉시컬 환경의 참조,

  즉 상위 스코프를 **"자신이 존재하는 한 기억한다."**

24.3 클로저와 렉시컬 환경

```
const x = 1;
```

```
// ①
function outer() {
  const x = 10;
  const inner = function () { console.log(x); }; // ②
  return inner;
}

// outer 함수를 호출하면 중첩 함수 inner를 반환한다.
// 그리고 outer 함수의 실행 컨텍스트는 실행 컨텍스트 스택에서 팝되어 제거된다.
const innerFunc = outer(); // ③
innerFunc(); // ④ 10
```

- 외부 함수보다 중첩 함수가 더 오래 유지되는 경우 중첩 함수는 이미 생명 주기가 종료한 외부 함수의 변수를 참조할 수 있다.
  - 이러한 중첩 함수를 클로저 라고 한다.
- 상위 예시에서, outer 함수의 실행 컨텍스트가 실행 컨텍스트 스택에서 제거되더라도 outer 함수의 렉시컬 환경은 소멸하지 않는다.
- inner 함수의 [[Environment]] 내부 슬롯에 의해 참조되고 있고, innter 함수는 전역 변수 innerFunc에 의해 참조되고 있으므로, 가비지 컬렉팅이 되지 않기 때문이다.
- 이론적으로 자바스크립트의 모든 함수는 상위 스코프를 기억하기 때문에 클로저
  - 하지만 상위 스코프의 어떠한 식별자도 참조하지 않는다면

    모던 브라우저에 의해 최적화되어 상위 스코프를 기억하지 않는다.
- 클로저는 중첩 함수가 상위 스코프의 식별자를 참조하고 있고 중첩 함수가 외부 함수보다

  더 오래 유지되는 경우에만 한중하는 것이 일반적.
- 자유 변수 (free variable)
  - 클로저에 의해 참조되는 상위 스코프의 변수
  - 클로저란 “함수가 자유 변수에 대해 닫혀있다.” 라고 표현 가능
    - 자유 변수에 묶여있는 함수

24.4 클로저의 활용

- 상태를 안전하게 은닉하고 특정 함수에게만 변경을 허용하기 위해 사용

```
const counter = (function () {
  // 카운트 상태 변수
  let num = 0;

  // 클로저인 메서드를 갖는 객체를 반환한다.
  // 객체 리터럴은 스코프를 만들지 않는다.
  // 따라서 아래 메서드들의 상위 스코프는 즉시 실행 함수의 렉시컬 환경이
다.
  return {
    // num: 0, // 프로퍼티는 public하므로 은닉되지 않는다.
    increase() {
      return ++num;
    },
    decrease() {
      return num > 0 ? --num : 0;
    }
  };
}());

console.log(counter.increase()); // 1
console.log(counter.increase()); // 2

console.log(counter.decrease()); // 1
console.log(counter.decrease()); // 0
```

```
const Counter = (function () {
  // ① 카운트 상태 변수
  let num = 0;

  function Counter() {
    // this.num = 0; // ② 프로퍼티는 public하므로 은닉되지 않는
다.
  }
```

```
  Counter.prototype.increase = function () {
    return ++num;
  };

  Counter.prototype.decrease = function () {
    return num > 0 ? --num : 0;
  };

  return Counter;
}());

const counter = new Counter();

console.log(counter.increase()); // 1
console.log(counter.increase()); // 2

console.log(counter.decrease()); // 1
console.log(counter.decrease()); // 0
```

- increase, decrease 함수는 모두 클로저
- 자신의 함수 정의가 평가되어 함수 객체가 될 때 실행 중인 실행 컨텍스트 (즉시 실행 함수)의 렉시컬 환경을 기억한다.
- 즉, num 변수는 increase, decrease 메서드만 변경할 수 있다.

```
// 함수를 인수로 전달받고 함수를 반환하는 고차 함수
// 이 함수는 카운트 상태를 유지하기 위한 자유 변수 counter를 기억하는 클로저를 반환한다.
function makeCounter(aux) {
  // 카운트 상태를 유지하기 위한 자유 변수
  let counter = 0;

  // 클로저를 반환
  return function () {
    // 인수로 전달 받은 보조 함수에 상태 변경을 위임한다.
    counter = aux(counter);
    return counter;
  };
```

```
}

// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// 함수로 함수를 생성한다.
// makeCounter 함수는 보조 함수를 인수로 전달받아 함수를 반환한다
const increaser = makeCounter(increase); // ①
console.log(increaser()); // 1
console.log(increaser()); // 2

// increaser 함수와는 별개의 독립된 렉시컬 환경을 갖기 때문에 카운터
상태가 연동하지 않는다.
const decreaser = makeCounter(decrease); // ②
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

```
// 함수를 반환하는 고차 함수
// 이 함수는 카운트 상태를 유지하기 위한 자유 변수 counter를 기억하는
클로저를 반환한다.
const counter = (function () {
  // 카운트 상태를 유지하기 위한 자유 변수
  let counter = 0;

  // 함수를 인수로 전달받는 클로저를 반환
  return function (aux) {
    // 인수로 전달 받은 보조 함수에 상태 변경을 위임한다.
    counter = aux(counter);
    return counter;
  };
}());
```

```
// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// 보조 함수를 전달하여 호출
console.log(counter(increase)); // 1
console.log(counter(increase)); // 2

// 자유 변수를 공유한다.
console.log(counter(decrease)); // 1
console.log(counter(decrease)); // 0
```

24.5 캡슐화와 정보 은닉

- 캡슐화 (Encapsulation)

  객체의 상태를 나타내는 프로퍼티와, 프로퍼티를 참조하고 조작할 수 있는 동작인 메서드를 하나로 묶는 것.
- 정보 은닉을과 객체간의 결합도를 낮추기 위해 사용한다.
- 자바스크립트는 public, privated, protected와 같은 접근 제한자를 제공하지 않는다.
- 따라서 자바스크립트의 모든 객체는 기본적으로 public하다.

- 다음의 예제에서 _age 변수는 내부 변수이므로 private하다.
- 하지만 Person 객체가 생성될 때 마다 중복으로 생성된다.

```
function Person(name, age) {
  this.name = name; // public
  let _age = age;   // private
```

```
  // 인스턴스 메서드
  this.sayHi = function () {
    console.log(`Hi! My name is ${this.name}. I am ${_age}.
`);
  };
}

const me = new Person('Lee', 20);
me.sayHi(); // Hi! My name is Lee. I am 20.
console.log(me.name); // Lee
console.log(me._age); // undefined

const you = new Person('Kim', 30);
you.sayHi(); // Hi! My name is Kim. I am 30.
console.log(you.name); // Kim
console.log(you._age); // undefined
```

- Person.prototype.sayHi 메서드는 즉시 실행 함수의 지역 변수 _age를 참조할 수 있는

  클로저이므로, 중복 생성 없이 private 변수에 접근이 가능하다.
- 하지만 여러 개의 인스턴스를 생성할 경우 변수의 상태가 유지되지 않는다.
  - 해당 메서드는 단 한번씩만 생성되는 클로저이기 때문

```
const Person = (function () {
  let _age = 0; // private

  // 생성자 함수
  function Person(name, age) {
    this.name = name; // public
    _age = age;
  }

  // 프로토타입 메서드
  Person.prototype.sayHi = function () {
    console.log(`Hi! My name is ${this.name}. I am ${_age}.
`);
  };
```

```
  // 생성자 함수를 반환
  return Person;
}());

const me = new Person('Lee', 20);
me.sayHi(); // Hi! My name is Lee. I am 20.
console.log(me.name); // Lee
console.log(me._age); // undefined

const you = new Person('Kim', 30);
you.sayHi(); // Hi! My name is Kim. I am 30.
console.log(you.name); // Kim
console.log(you._age); // undefined
```

- 이처럼 자바스크립트는 정보 은닉을 완벽하게 지원하지 않는다.

24.6 자주 발생하는 실수

- 다음의 예제에서 for문 안에 있는 var i 변수는 함수 스코프만을 따르기 때문에 전역 변수이다.
- 따라서 funcs 배열 안에 저장된 3개의 함수는 모두 3을 반환한다.

```
var funcs = [];

for (var i = 0; i < 3; i++) {
  funcs[i] = function () { return i; }; // ①
}

for (var j = 0; j < funcs.length; j++) {
  console.log(funcs[j]()); // ②
}
```

- 클로저를 활용하면 다음과 같이 수정할 수 있다.
- 즉시 실행 함수가 반환한 중첩 함수는 자신의 상위 스코프를 기억하는 클로저이다.

- 매개변수 id는 즉시 실행 함수가 반환한 중첩 함수에 묶여있는 자유 변수가 되어 그 값이 유지

```
var funcs = [];

for (var i = 0; i < 3; i++){
  funcs[i] = (function (id) { // ①
    return function () {
      return id;
    };
  }(i));
}

for (var j = 0; j < funcs.length; j++) {
  console.log(funcs[j]());
}
```

- const, let 키워드를 사용하면 더 깔끔하게 해결할 수 있다.
- const, let 변수는 블록 레벨 스코프를 따르기 때문에 반복문이 돌 때마다 새로운 렉시컬 스코프를 생성한다.

```
const funcs = [];

for (let i = 0; i < 3; i++) {
  funcs[i] = function () { return i; };
}

for (let i = 0; i < funcs.length; i++) {
  console.log(funcs[i]()); // 0 1 2
}
```

- 또는 함수형 프로그래밍 기법인 고차 함수를 사용할 수 있다.

```
// 요소가 3개인 배열을 생성하고 배열의 인덱스를 반환하는 함수를 요소로 추가한다.
// 배열의 요소로 추가된 함수들은 모두 클로저다.
const funcs = Array.from(new Array(3), (_, i) => () => i);
// (3) [f, f, f]
```

```
// 배열의 요소로 추가된 함수 들을 순차적으로 호출한다.
funcs.forEach(f => console.log(f())); // 0 1 2
```